Programming Paradigms
Lecture 12:
Subroutines and Control Abstraction (Part 1)

Prof. Dr. Michael Pradel
Software Lab, University of Stuttgart
Winter 2019/2020
What does the following C code print?

```c
#include <stdio.h>

int *foo() {
    int a = 5;
    return &a;
}

int main() {
    int *p = foo();
    printf("%p\n", p);
}
```
Wake-up Exercise

What does the following C code print?

```c
#include <stdio.h>

int *foo() {
    int a = 5;
    return &a;
}

int main() {
    int* p = foo();
    printf("%p\n", p);
}
```

Result on my machine: (nil)

In general:
Undefined behavior
Wake-up Exercise

What does the following C code print?

```c
#include <stdio.h>

int *foo() {
    int a = 5;
    return &a;
}

int main() {
    int* p = foo();
    printf("%p\n", p);
}
```

Result on my machine: (nil)

In general:

Undefined behavior

Access to local variable of `foo`, which doesn’t exist anymore
Control vs. Data Abstraction

- Abstract a well-defined operation
  - E.g., a subroutine or an exception handler

- Abstract how to represent information
  - E.g., types and classes
Control vs. Data Abstraction

- Abstract a well-defined operation
  - E.g., a subroutine or an exception handler

- Abstract how to represent information
  - E.g., types and classes

Focus of this and next lecture
Terminology

- **Subroutine**: Mechanism for control abstraction
  - Function: Subroutine that returns a value
  - Procedure: Subroutine that doesn’t return a value
- **Parameters**
  - Actual parameters = arguments: Data passed by caller
  - Formal parameters: Data received by callee
Overview

- Calling Sequences
- Parameter Passing
- Exception Handling
- Coroutines
- Events
Calling Sequences

- Low-level code executed to maintain call stack
  - Before subroutine call in caller
  - At beginning of subroutine in callee ("prologue")
  - At end of subroutine in callee ("epilogue")
  - After subroutine call in caller
Why Does It Matter?

- Important to
  - Understand performance implications
  - Understand security implications, e.g., stack smashing attacks
  - Choose/design/implement compilers
Reminder: Stack Layout

- Each procedure call:
  One stack frame (or activation record)
- Frame pointer: Base address used to access data in current stack frame
- Stack pointer: First unused (or, sometimes, last used) location in current stack frame
Example: Stack Layout

Stack pointer

Frame pointer

Current frame

Variables etc.

Previous (calling) frame

Direction of growth (towards lower addresses)
Tasks to Perform

- Pass parameters and return value(s)
- Update program counter
- Save return address
- Save and restore registers
- Update stack and frame pointers
Tasks to Perform

- Pass parameters and return value(s)
- Update program counter
- Save return address
- Save and restore registers
- Update stack and frame pointers

Program counter: Address of code to execute next
Tasks to Perform

- Pass **parameters and return value(s)**
- Update **program counter**
- Save **return address**
- Save and restore **registers**
- Update **stack and frame pointers**

Otherwise, don’t know what code location to return back to
Tasks to Perform

- Pass parameters and return value(s)
- Update program counter
- Save return address
- Save and restore registers
- Update stack and frame pointers

Registers: Very fast but limited intermediate memory
Tasks to Perform

- Pass parameters and return value(s)
- Update program counter
- Save return address
- Save and restore registers
- Update stack and frame pointers

Where to perform those?

- Possibly either in caller or in callee
- Preferably in callee: Requires space only once per subroutine, not at each call site
Typical Calling Sequence (1/4)

- Steps performed by caller before the call
  - Save registers whose values may be needed after the call
  - Compute values of arguments and move them into stack or registers
  - Pass return address and jump to subroutine
Stack before call

Stack pointer (off caller)

Return address

Arg 1
Arg 2
...

Caller frame

Calling frame
Steps performed by callee in prologue

- Allocate a frame: Subtract an appropriate constant from the stack pointer
- Save old frame pointer on stack and update it to point to newly allocated frame
- Save registers that may be overwritten by current subroutine
Stack after prologue

Stack pointer

Frame pointer

Caller frame

Saved registers

Saved frame pointer

Return address

Arg 1

Arg 2

Calling frame

Direction of growth

Higher addresses
Quiz: Stack Frames

Assume the frame pointer is stored in register $ebp$, addresses are 4 bytes long, and all arguments are 32-bit integers.

What is the address the callee uses to access the second argument?
Quiz: Stack Frames

Assume the frame pointer is stored in register `ebp`, addresses are 4 bytes long, and all arguments are 32-bit integers.

What is the address the callee uses to access the second argument?

Answer: `ebp + 12` bytes
Typical Calling Sequence (3/4)

- **Steps performed by callee in epilogue**
  - Move *return value* into register or reserved location in stack
  - Restore registers (to state before call)
  - Restore frame pointer and stack pointer
  - Jump back to return address
Stack after epilogue:

- Arg 1
- Arg 2

Local variables:
- Saved registers
- Saved frame pointer
- Return address

Direction of growth:
- Higher addresses
Typical Calling Sequence (4/4)

- Steps performed by caller after the call
  - Move *return value* to where it is needed
  - Restore *registers* (to state before call)
Saving and Restoring Registers

- Which registers to save and restore?
  - E.g., x86 has 8 general purpose registers
  - Ideally, save if
    - Caller may use them after the call
    - Callee needs them for other purposes
  - In practice, compiler safely overapproximates
    - Better to store once too much then to loose data
Saving and Restoring Registers

Where to save and restore registers?

- Caller could save (and then restore) all registers that are in use
- Callee could save (and then restore) all registers it overwrites
- In practice: Calling sequence conventions for each architecture
  - E.g., on x86, three registers are caller-saved, rest is callee-saved
Inlining

- Calling sequences are expensive
- Optimization, in particular for small functions:
- **Inlining**
  - Copy of callee becomes part of caller
  - Avoids overhead of calling sequence
  - Enables other optimizations across subroutine boundaries
  - But: Increases code size
Example: Inlining Hints in C

- Programmer may suggest which subroutine to inline
- Example:

```c
inline int max(int a, int b) {
    return a > b ? a : b;
}
```
Application: Stack Smashing

- Special kind of buffer overflow vulnerability
  - Lack of bounds checking: May write beyond space allocated for a local variable
  - Malicious input can overwrite return address
  - Program will jump into malicious code
Example: Stack Smashing

```c
int read_nb_from_file(FILE *s) {
    char buf[100];
    char *p = buf;
    do {
        /* read from stream s */
        *p = getc(s);
    } while (*p++ != '\n');
    *p = '\0';
    return atoi(buf);
}
```
Example: Stack Smashing

...      stack growth

buf

...      higher addresses

return address
Overview

- Calling Sequences
- Parameter Passing
- Exception Handling
- Coroutines
- Events
Parameter Passing

- What does it mean if a parameter is passed to a callee?
- Different PLs have different parameter passing modes
  - Call by value
  - Call by reference
  - Call by value/result
  - Call by sharing
Call by Value

- Caller passes copy of value to callee
- Once in callee, formal parameter is independent of the actual parameter
Call by Value

- Caller **passes copy of value** to callee
- Once in callee, formal parameter is independent of the actual parameter

```plaintext
x : integer
procedure foo(y : integer)
  y := 3
  print x
...
x := 2
foo(x)
print(x)
```
Call by Value

- Caller **passes copy of value** to callee
- Once in callee, formal parameter is independent of the actual parameter

```verbatim
x : integer
procedure foo(y : integer)
  y := 3
  print x
...
```

Assignment has no visible effect

```verbatim
x := 2
foo(x)
print(x)
```
Call by Value

- Caller **passes copy of value** to callee
- Once in callee, formal parameter is independent of the actual parameter

```plaintext
x : integer
procedure foo(y : integer)
  y := 3
  print x
...
x := 2
foo(x)
print(x)
```

Prints 2 twice
Call by Reference

- Formal parameter is new name of actual parameter
- Both names refer to the same value
Call by Reference

- Formal parameter is **new name of actual parameter**
- Both names **refer to the same value**

```plaintext
x : integer
procedure foo(y : integer)
  y := 3
  print x
...
x := 2
foo(x)
print(x)
```
Call by Reference

- Formal parameter is **new name of actual parameter**
- Both names **refer to the same value**

```plaintext
x : integer
procedure foo(y : integer)
    y := 3
    print x
...
x := 2
foo(x)
print(x)
```

Refers to same value as outer variable `x`
Call by Reference

- Formal parameter is **new name of actual parameter**
- Both names **refer to the same value**

```plaintext
x : integer
procedure foo(y : integer)
    y := 3
    print x
...
x := 2
foo(x)
print(x)
```

Prints 3 twice
Call by Value/Result

- Argument is copied on call
- Resulting value is copied back to argument on return
Call by Value/Result

- Argument is copied on call
- Resulting value is copied back to argument on return

```plaintext
x : integer
procedure foo(y : integer)
    y := 3
    print x
    ...
    x := 2
foo(x)
print(x)
```
Call by Value/Result

- Argument is copied on call
- Resulting value is copied back to argument on return

```pascal
x : integer
procedure foo(y : integer)
  y := 3
  print x
...
x := 2
foo(x)
print(x)
```

Quiz: What does the code print?
Call by Value/Result

- Argument is copied on call
- Resulting value is copied back to argument on return

```plaintext
x : integer
procedure foo(y : integer)
    y := 3
    print x
...
x := 2
foo(x)
print(x)
```

Refers to value different from outer variable x
Call by Value/Result

- Argument is copied on call
- Resulting value is copied back to argument on return

```plaintext
x : integer
procedure foo(y : integer)
    y := 3
    print x
...
x := 2
foo(x)
print(x)
```

Prints 2
Prints 3
Call by Sharing

- In PLs with reference model of variables
  - Arguments are passed as values, but the values are references
Call by Sharing

- In PLs with reference model of variables
  - Arguments are passed as values, but the values are references

```plaintext
x : integer
procedure foo(y : integer)
  y := 3
  print x
...
x := 2
foo(x)
print(x)
```

Prints 3 twice
Passing Models in Popular PLs

- **C**: By value, except arrays are passed by reference
  - Passing pointers can emulate call by reference
- **Fortran**: All arguments are passed by reference
- **Java**: Hybrid passing model
  - Built-in, primitive types: By value
  - Instances of classes: By reference
Overview

- Calling Sequences
- Parameter Passing
- Exception Handling
- Coroutines
- Events